Security in the Sanctuary System*

Matthew Hohlfeld† Aditya Ojha‡ Bennet Yee§

September 20, 2002

Abstract

The Sanctuary mobile code system includes security mechanisms for protecting mobile agents from malicious servers as well as mechanisms for protecting mobile agent servers from malicious mobile code.

To protect remotely executed mobile code, we integrate several key approaches: (1) security attributes certification to enable mobile code to avoid nodes in the agent-server network that are untrustworthy, as determined by user-centric security policies; (2) forward secure cryptography to improve detection of malicious tampering by servers; and (3) defining separate roles for agent author and agent owner, which justifies restricted delegation and external reference monitors with owner-provided agents to limit potential damage caused by buggy or compromised agent code. Simply put, we enable mobile code to avoid trouble when possible, and to detect trouble when it is unavoidable. We examine security-aware itinerary planning as a means to supplement these approaches, and describe our analysis of this problem. Our server uses well known approaches to defend itself from malicious code, and custom extensions that address the security needs of the mobile code itself. This paper describes our mechanisms and how they are integrated into the Sanctuary mobile code system.

1 Introduction

The Sanctuary project is investigating the security limitations of mobile agent systems. We are motivated by the ability of mobile code to autonomously control its execution location. Explicit location control enables properly written software to eliminate much of the communication latency between the computation and its

*This research was funded in part by the National Science Foundation, CAREER Award CCR-9734243; and the Office of Naval Research, Award N00014-01-1-0981.

†`hohlfeld@cs.ucsd.edu`

‡`aojha@cs.ucsd.edu`

§`bsy@cs.ucsd.edu`

needed external resources [14], and allows mobile code systems to outperform traditional RPC-based systems in high-latency environments. In this paper, we give an overview of the Sanctuary system architecture and discuss in greater depth our security mechanisms: (1) attribute certificates containing security evaluation results to enable the use of user-supplied risk management decision functions; (2) forward secure logging of partial results to detect single failures (described elsewhere [40]); and (3) service request interposition to allow stackable restrictions for restricted delegation of user authority, resource usage control, and rights amplification.

We believe that structuring distributed applications using mobile code is an important way to adapt to trends in hardware technology. In the next section, we briefly motivate the need for mobile code and the criticality of security for mobile code. We discuss our viewpoint on agent security goals and briefly describe mechanisms used to provide agent security in Section 3. We describe our system system architecture and security-specific design and implementation details in Section 4.

2 Motivation

In the Sanctuary project, we focus on the security of remote code execution. We view that performance needs, especially in the face of inevitable technology trends, will make the ability to securely execute code remotely critical for distributed systems performance.

Code mobility solves a critical problem: it eliminates most of the communication latency, collapsing multiple rounds of communications to one by co-locating the code with remote resources. Suppose an application needs to access a remote resource repeatedly, conventionally using RPCs. The communication latency can easily dominate the computation time. By restructuring the application to use mobile code, we may end up using more system resources, but we need only pay for one network round-trip, greatly improving the time-to-completion. For applications where time-to-completion is a more important metric than overall resource utilization efficiency, mobile code is very attractive.

Furthermore, available computation power and communications bandwidth among distributed nodes have been increasing at exponential rates, albeit the doubling times differ. While these technology trends allow us to solve ever larger problems, they also lead to an inescapable conclusion: distributed systems performance is moving inexorably towards a *latency dominated regime*. Absent new physics, communications latency cannot decrease beyond the simple physical limits imposed by physical separation and the signal propagation speed.

Some distributed applications are already clearly latency dominated. Let us look at an extreme case as a point of comparison. Planetary robotics must make decisions locally whenever possible: the 1997 Pathfinder mission [33] performed image analysis and path planning using a local 20 MIPS processor rather than suffering a message roundtrip of 20–40 minutes in an RPC to much faster Earth-bound server processors. Here, waiting for the reply would have an opportunity cost of 24×10^9 local instructions, a clearly unacceptable alternative. Consider now an Earth-bound transoceanic RPC. A back-of-the-envelope calculation shows that a similar instruction-count latency penalty will occur for RPCs using desktop-class hardware expected to be available in less than 9 years! Indeed, some conventional distributed applications are already latency dominated. NFS, for example, has little hope of performing well across high latency networks, especially in the face of write sharing [34, 35]. Soon, *all* communications links will be high latency when compared with local processing.

To realize the gains from hardware performance improvements, distributed applications must be restructured as more applications move into the latency dominated regime. Design techniques such as client-server interactions with RPCs tend to result in systems that require many message roundtrips, and application performance concomitantly suffers. And while resource utilization and throughput can be enhanced by simply context switching to exploit inter-job parallelism and avoid idling resources, such an approach does nothing to reduce the time-to-completion for individual jobs and can only exacerbate the situation.

The anticipated ubiquity of remotely executing code, however, raises a plethora of security concerns. Servers must defend themselves against malicious mobile code. Similarly, users of mobile-code enabled applications will not necessarily trust each other nor the administrators of the remote machines upon which their code may run, and their mobile agents must be protected from malicious servers and other malicious agents. This paper describes mechanisms in the Sanctuary mobile code system that address these agent security issues.

3 Agent Security Concepts

The Sanctuary System addresses some new goals for agent security, as well as refining standard security goals and examining the interactions of security mechanisms to provide an integrated approach to agent system security. These new security goals require us to provide mechanisms that address attacks that aren't meaningful in other systems.

Our system includes mechanisms to address the standard agent security goals, described elsewhere [15]. A wide array of previous work has created security models for and described attacks on agent systems in general [14] and for particular agent systems such as Aglets [17], JavaSeal [8] and SeMoA [31]. This section describes some of the concepts we use while designing mechanisms for our system.

3.1 Overview of Goals and Mechanisms

Each refinement on mobile agent security models is an attempt to make a model that better reflects the security needs of mobile agent programmers, users and agent system administrators. To remain practical, the new model must remain achievable through realistic security mechanisms.

Mobile agent systemss need not be deployed with defenses against all possible attacks to be a generally useful tool. We need only identify and defend against those attacks ultimately damaging to individual applications and servers. Along with our contributions, recent work in this field provides sufficient security mechanisms that reasonable applications can be developed and deployed in a secure manner.

Our system uses standard mechanisms for isolating mobile agents, protecting mobile agents from agent servers and vice-versa. In addition to these standard mechanisms, we provide mechanisms to achieve new goals for agent functionality and security. Our methods partially protect the integrity of mobile agents' computation and data, and we provide additional mechanisms for protecting privacy.

One refinement that we have made to the general agent programming model is to consider the case where mobile code that is part of an agent may not be trusted by the agent's owner. The *agent owner* for an agent is the entity that configures then starts the agent. The agent owner accepts responsibility for the agent's actions by delegating some of the owner's rights to it. A running mobile agent may be composed of off-the-shelf code from multiple sources, none of which are completely trusted by the agent owner. In this case it is important that the agent owner be able to control the agent as it executes and ensure that it doesn't behave in a manner contrary to the agent owner's stated policy. This refinement on the model is reflected in our goal to provide a mechanism for restricting rights delegations.

We now highlight the unique goals in our design that provide the motivations for our security mechanisms:

- **Delegation of rights to agents:** Account-based authorization systems clearly do not scale to global systems. Therefore, agents in a global agent system must be able to acquire rights without requiring a prior account relationship between the agent owner and each agent server. Additionally, agents cannot securely hide keys from the servers that they execute on. Therefore, we must provide a means for delegating rights to individual agents without first binding them to a cryptographic key. We delegate rights through delegation certificates that bind those rights to an agent's "natural name", which is a strong form of agent identity. Agent identity is described in detail in Section 4.2. In addition to allowing us to bind rights to agents, the globally unique nature of the natural name enables us to dynamically allocate resources to an agent and make them available for the agent even if it migrates out of the allocating server and then later returns to it. We use an agent's natural name to provide it with secure access to its short-term cryptographic keys across migrations.

- **Restriction on rights delegations:** Rights delegated to an agent need not be delegated without conditions. As described above, an agent owner need not completely trust the mobile code that they use in their agent. When an agent is created from untrusted components it is important that the agent owner be able to specify which limitations to impose on rights delegated to the agent.

  We implement restrictions on delegated rights with programmable interposition agents, described in Section 4.9. These interposition agents intercept all communications between the untrusted mobile code and the service examining the delegated rights. This structure allows the interposition agent to modify or deny requests before the service provider sees them, and can also be used for extended features such as logging or manipulating the responses to those requests.

- **Secure logging of partial results:** The results of an agent's computation must be protected from dishonest servers on its itinerary in order to provide a partial guarantee about the integrity of its computation. This is achieved in our system by securely maintaining logs of the partial results computed on each server. Forward secure cryptographic support (see Section 3.2) is used by the SDR logging module for maintaining such a log, as described in Section 4.7.

- **Access to security-relevant information:** We believe that dynamic security information about the parties involved should be available to programs so that access control decisions and migration itineraries need not rely completely on the information available prior to execution. Furthermore, both agents and servers should be able to make use of this dynamic security information.

  To provide dynamic security information in our system, we extend the general concept of attribute certificates to include security attributes (Sections 3.4 and 4.5).

- **Secure itineraries:** To ensure that their results are computed from the correct inputs and their actions are performed correctly, mobile agents must minimally have a means for ensuring that the sequence of servers they visit matches the itinerary they planned to visit.

  Without external support beyond the standard use of server-provided authenticated links, it is only possible to ensure that the portion of the itinerary before visiting a malicious server and after the last malicious server on the itinerary. With the introduction of monotonic server variables [38], we can additionally ensure that the sequence of honest servers between any pair of colluding malicious servers on the itinerary will only be traversed once.

More robust mechanisms which require agent servers to sign statements about the agent's execution path can be added at the agent level, and will allow a verifier to determine whether or not all of the intended hosts were on the path taken by the agent, and in the correct order. We will not discuss the mechanisms for providing authenticated links and signature-based path verification here.

In our model, single migrations have further security requirements as described in Section 3.3. We discuss the planning of secure itineraries in Section 4.8.

Our approaches to and solutions for some standard goals in the design of secure agent systems have direct effects on the design of several of the mechanisms that address the above goals. We briefly describe these goals here:

- **Safe execution of agents:** Though only limited claims can be made about the protections given to an agent against malicious servers, honest servers can provide significant protection between agents. By keeping an agent free from tampering by other agents, we simplify both the programming task for agent programmers and the model for analyzing their security.

  We use standard mechanisms for isolating agents from each other when running in a Java-based mobile agent server. We chose to use strict separation of agent object graphs, restricting agent interaction to a single read-write interface (the port communication interface, see Section 4). This separation provides the opportunity for a clean implementation of the interposition mechanism on both inter-agent interactions and agent-server interactions.

- **Simplicity of programming model:** We provide only basic security and functionality in the server itself, guided in its design by a micro-kernel model. In order to provide a simple programming model to agent authors, we provide hooks in the infrastructure for migration-aware libraries. These libraries operate at the agent level and provide high-level interfaces to agent programmers.

  The full design for including migration-aware libraries in our system and the high-level code mobility interface provided by the Mojo source-to-source precompiler are presented in other work [10, 15].

3.2 Forward Security for Agents

We would like to ensure two security properties for an agent's computation: forward integrity and forward confidentiality [2, 20, 39]. Our approach utilizes existing techniques in forward secure cryptography to achieve these goals. Typically, forward secure cryptography is used to ensure that security properties arising from cryptographic operations performed in earlier time periods cannot be violated even if cryptographic secrets for the current time period are compromised. This is done by updating the cryptographic secrets in a one-way manner at the end of each time period. Our use of the concept differs slightly in that the earlier time periods for a mobile agent correspond to its execution on earlier servers on its itinerary.

The Sanctuary system includes a secure partial result logging library for agents that employs two existing forward secure cryptography schemes. The first scheme uses a forward secure pseudo-random number generator, a Message Authentication Code (MAC) and a symmetric encryption scheme to achieve both forward security properties [5]. The second scheme provides forward integrity using a forward secure, public key based, digital signature scheme [4]. The implementation of the SDR library is discussed in Section 4.7.

3.3 Nested Transactions through Parallel Protocols

We introduce the concept of using a coordinated set of parallel protocols to implement a restricted form of nested transactions [19, 21]. The restriction we require does not allow for fully general nested transactions: only a single layer of nesting is allowed. The set of protocols is viewed as a single transaction, with each protocol implementing its own transaction nested in the set. Consistent with the nested transaction model, if any of the individual protocols (or transactions) aborts, the entire transaction must abort, and similarly for committing.

By implementing the agent migration protocol as one such nested transaction, we allow for extensions to it while restricting the possible failure modes from adding new protocols. The mechanism for extensible agent migration protocols is shown in Section 4.4.

3.4 Security Attributes

In addition to the access restrictions and resource needs of a particular agent or the current availability of those resources on a server, agents and servers may need to make decisions based on the security attributes of the parties with which they communicate. The *security attributes* of an entity include all security-relevant information and statements about it. These attributes may serve as inputs to another entity's security decisions governing the scope of their interactions. For example, whether a server has undergone penetration testing or other forms of security audit—and the results of those tests as well as the identity of the agency that conducted them—would be useful for mobile agents and their users since mobile agents can use this information to avoid untrustworthy servers.

Server administrators and agent programmers will trust different authorities to verify the security attributes of other servers and agents. To satisfy these disparate trust relationships, we must provide a decentralized mechanism for allowing authorities to securely make statements about security attributes and provide them to the active parties in the system. This is the motivation for our security attribute certificates as described in Section 4.5.

4 Security Components

The Sanctuary Agent SErver (SASE) is a Java application that provides a low-level interface for building distributed and mobile middle-ware and applications. We use a micro-kernel approach [1] in the SASE, providing only minimal agent creation and communication mechanisms in the server itself. Agent libraries and special agents called *service agents* reside in a server to provide extended services to agents and to provide access to protected resources.

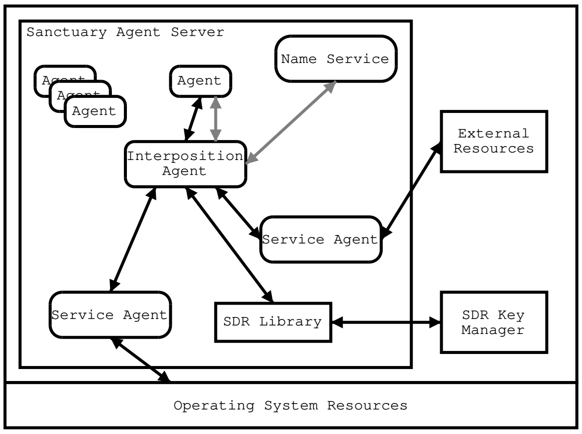

Figure 1: Server Architecture

Each server contains several subsystems, and may be running several agents at once.

In the SASE, running mobile agents are isolated from each other, though we neither use as strict a hierarchy as the seals in JavaSeal [8], nor as loose a structure as the contexts of Aglets [18] or the thread groups of SeMoA [31]. We chose to provide a strict object-graph separation similar to that in JavaSeal while allowing the more dynamic interaction pattern of other systems.

The SASE provides *ports*, which are access-controlled single-receiver message queues that allow agents to communicate with each other and with the various service agents resident on the server. The design for the port communication mechanism is based on the Mach IPC design [26, 36]. The port implementation maintains the separation of agent object graphs by serializing (and de-serializing) all objects passed through them. We indicate a pair of ports in our figures by double-headed arrows between communicating entities in the system.

The Security Attribute Certification Infrastructure (SACI) allows agents and servers to use security attribute certificates to prevent problems before they occur by first examining their trust of the planned action. The Secret Decoder Ring (SDR) mobile code support library helps agent users detect problems in the execution of their mobile code.

4.1 Security Mechanisms for Agents

Mobile agents must rely on the agent server to provide some of the mechanisms for their security. Other mechanisms can be implemented at the agent level: as library code invoked by the agent or through services provided by other agents running in the same system. We will briefly show which components reside in each category.

4.1.1 Mechanisms Provided by the Server

Fundamentally, the agent server itself and all of its underlying software must behave properly to ensure a correct execution environment for mobile agents. The agent server uses standard agent isolation techniques that require a unique class loader for each agent, and prevents object reference sharing between agents. To maintain these distinct object graphs, the agent server requires agents to communicate through the port communication mechanism. The port manager is responsible for creating ports and securely identifying parties engaged in communication. Inter-server communication, specifically the agent creation (and migration) mechanism, is secured by the server's use of bi-directionally authenticated SSL connections managed by its communications module. Finally, extended security features are provided by libraries loaded by the server such as the forward secure cryptographic routines provided by the SDR library.

4.1.2 Mechanisms Provided at the Agent Level

Agent programmers can extend the agent server-provided security to ensure additional security services. At the agent-level, some standard services include: interposition agents for actively restricting the use of delegated rights, a controlled name space for advertising services, and the logging component of the SDR library.

4.2 Agent Identity

Without a secure mechanism for identifying agents, cryptographic methods to protect agents from malicious servers break down [29].

We provide such a mechanism for identifying agents. In a manner similar to the SPKI/SDSI concept that "the name is the key", we use "natural names" to refer to mobile agents. The *natural name* for a mobile agent is the composition of the portion of the agent that doesn't change during its execution. The static portion of an agent's state includes all of the configuration data provided when the agent is intialized: the basic code for the agent, the read-only portion of the agent's object graph, the agent's configuration (including its static policy), the agent owner's public key and an instance number unique to that agent owner, as shown in Figure 2. This extends the sketch of an agent's *kernel* [30], which requires that it be comprised of "all its data, code, and configuration information that does not change during the agent's lifetime" to describe which components are minimally required in our system to achieve a secure, globally unique, agent identity. Because agents may use dynamically bound (or generated) code, we do not assume that all code that the agent uses is included in this kernel, but rather that the code to verify that the binding (or generation) is correct is there.

This name is inconveniently large for normal use, however, and in practice we use a cryptographically secure hash of the full name to act as a manageable short version of the agent's natural name.

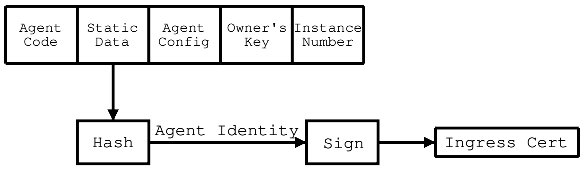

Figure 2: Agent Identity

Agent identity is created implicitly from the agent's configuration.

The agent owners will use this short version in certificates claiming ownership of the agent, thereby tightly binding the agent (and its configuration) to them. This ownership is a form of indemnification for the agent, such that agent servers can rely on the bound identity (or identities) for actions taken by the agent on the owners' behalf. Though agents are typically referred to as having a unique agent owner, in practice multiple entities can accept responsibility for an agents actions, thereby providing it with access to a greater number of servers and resources.

4.3 Agent Groups

We generally refer to each mobile agent as a single entity, however, our model allows an agent owner to create a mobile agent as an *agent group*, or set of agents running and migrating in concert. The individual agents in an agent group are called *member agents*. Agents structured this way can carry some service agents with them, yet interact with them using standard port communication.

We now consider an example agent group composed of three member agents. This agent group contains the member agents and communication pattern seen in Figure 3. The member agents are defined as follows: the (configurable) store-bought agent generates the set of indexing queries that the agent owner wishes to see, the database agent is authorized by the database resource owner to provide (controlled) access to the database, and the index agent uses a proprietary algorithm to access an index created from information in the database.

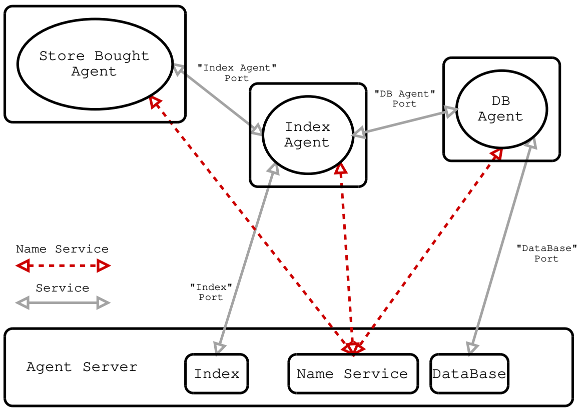

Figure 3: Agent Groups

Agent groups are composed of multiple member agents.

When an agent is structured as an agent group, the agent's natural name is slightly different. The static portion of the agent's state is now composed of the static portion of the agent group's state. This, in turn, is composed of the natural names for the member agents. This is a clear extension of the agent identity described above, and will not be discussed further.

4.4 Protocol Bundles

In an optimized agent itinerary, agent migration is the only remaining high latency communication on the agent's critical path of execution. Agent migration must be implemented with the minimum number of communication round-trips to ensure minimal impact on time-to-completion.

The SASE includes support for dynamically extensible agent migration protocols through protocol bundles. A *protocol bundle* is a set of coordinated parallel protocols that implements the semantics of a single nested transaction. Additional protocols are added to a protocol bundle to support services that react to an agent migration with their own protocols. Forward-secure key transfer, as seen below in Section 4.7.3, is one such protocol. These protocols run in parallel with the agent server's built-in agent transfer protocol, whose sole duty is transferring agent data for the low-level agent creation operation.

Though each protocol is provided with information about the overall transaction, it is isolated from the other sub-protocols, and can only interact with its peers through the protocol bundle itself. This design allows sub-protocol authors to write their protocols independently, with only a minimum of trust that other sub-protocols behave properly.

Unfortunately, the loose synchronization between protocols in our design allows for leakage of sensitive information to the remote host. This can only occur in the event of a migration failure, however, which implies that the agent already trusted the remote server enough to attempt a migration to it. The information leaked to the remote server is thus in the control of a partially trusted server and is not a severe security problem.

To ensure properties outside of the main-line execution of the agent's code across agent migrations, we call hook functions on each migration success or failure. This design is similar to the Aglet system's [18] `onArrival` callback. An agent author may choose, for example, to use state appraisal functions [12] whenever the agent successfully migrates to a remote host, or to reclaim references to non-serializable objects that it would have lost during the agent migration.

4.5 Security Attribute Certificates

We use Security Attribute Certificates (SACs) to distribute security attribute information in a decentralized manner. Whereas traditional certificates bind identity to keys and delegate rights to key holders, we extend attribute certificates [22] in SACs to bind security attributes of a key holder to that key. As described in Section 4.2, the "key" used in a SAC that refers to an agent is the agent's natural name. SACs are issued by Security Attribute Certification Authorities (SACAs), which are analogous to the traditional Certification Authorities (CAs) that issue identity certificates.

The Security Attribute Certification Infrastructure (SACI) defines the role of the SACA and includes a complete description of the SAC design. SACI extends and retargets the SPKI/SDSI [11, 28] certificate design into a mobile code context.

SACs act as a form of "secured input data" from the SACA to the user of the certificate. Rather than using a certificate revocation list, we use recency requirements and expiration dates to control the lifetime of SACs [27]. The meaningful lifetime of security attribute certificates can be related to the security attributes that they contain: operational security attributes (such as mechanisms for disaster recovery, backups and personnel security) will have long lifetimes: the attributes and our understanding of them change very slowly; individual attributes for a particular system or piece of software that relate to ongoing analysis of that entity will have short lifetimes: penetration testing of systems and security analysis of code may cause rapid changes in our understanding of their security.

We need not require that the information in SACs immediately reflect changes in the current state of knowledge about a particular entity. We must only ensure that unacceptable security vulnerabilities do not arise from the difference between what is known and what is represented as true in the SACs. In particular, expiry of SACs that are bound to a piece of software due to security-relevant bugs found in the software can happen at a human time-scale. Given that current efforts for finding security-relevant bugs are largely performed by people, both finding and exploiting these bugs happen on a human time-sscle. Thus, certificate lifetimes (and thus certificate expiry) on the scale of a day or more are quite reasonable when a security compromise requires non-trivial human effort. This conclusion will not hold true when either the bug can be exploited automatically (such as automatically generated exploit code for previously identified buffer overflows), or when the security exposure due to inaccuracy in the certificate corresponds to significant potential damage.

No agent systems offer protection against physical attacks on the agent servers or attacks on configuration management failures for trusted components. SACs allow us to encode information about these types of attacks so security policies can use them to take susceptibilty to these attacks into account.

4.6 Policies and Policy Decisions

Each agent and agent server may have its own policy for application and security decisions. This policy, when input to a policy engine along with supporting evidence, determines which actions will be taken or denied using a trust management-based policy mechanism [7, 9, 6]. Access control mechanisms refer to the policy engine to determine adherence to the rights delegation rules and the policy assigned to their resources.

Policy decisions are made based on information from multiple sources, including both dynamic information, such as server usage, and static information such as certificates and policies. The decisions controlled in this manner differ by situation and entity.

Policies that use SACs as input are required to specify not just which attributes to examine from SACs, but also which of the trusted authorities is allowed to assign each attribute. We introduce this limitation on Certification Authorities (CAs) since the role a SACA plays in a policy depends upon the type of certificates it is allowed to issue. When CAs are only used for certifying identity, there is no need to differentiate between them beyond whether or not they are trusted to certify identities.

A policy may be extended by properly formed policy updates received while the user of the policy is running. Thus an agent creator, server administrator, or other duly delegated entity, may create a new policy that will have the same force as the original policy when those updates are incorporated in it. This update mechanism allows policies (and the programs that they are used by) to remain valid as new security information is obtained.

Section 4.9 showed us how interposition agents could be used to implement agent policies for interaction with other agents. The agent owner policy interpreted by a policy engine would provide sufficient information to agent library code that the concerns about interposition-based policy enforcement described in Section 4.9.2 would be addressed. The primary difficulty with the interposition mechanism for policy enforcement is that the lack of communication between layers implies that a simple layering approach is too strict, as we expect different policies to interact, thereby requiring that the enforcement of the policies cooperate. By providing both a policy engine for interpreting and storing policies and a separate enforcement mechanism via interposition agents, we allow a variety of different approaches to agent and server policies that covers reasonable alternatives for incorporating policies into the mobile agent model.

4.7 Forward Secure Partial Result Logging

Agents can protect their partial results using the forward secure cryptography schemes provided by the SDR library. The SDR library consists mainly of two modules: a high-level result logging and verification library which invokes routines in a low-level module that implements the cryptography schemes.

4.7.1 Issues with Maintaining Forward Security

The SDR library must interface with Java programs—the SASE and the agents running on it. Although the SDR result logging routines are implemented in Java, the cryptography routines are not. The cryptographic key handling functions and the actual cryptography schemes are implemented in native code. Cryptographic secrets stored in Java objects cannot be reliably erased because Java does not provide the necessary controls over memory management. In particular, copies of secrets contained in Java objects may remain on a server due to garbage collection and paging. Thus, the keys and scratch space used in the forward secure cryptography schemes cannot be stored in Java objects. The cryptography routines are implemented as a JNI library written in C, which allows explicit control over memory allocation and pinning of memory pages to main memory.

Another issue that complicates the implementation is transfer of the secrets that are used in the forward secure cryptography schemes to the next server. Since time periods in the cryptography schemes change when an agent migration begins, the secrets that are used on a server must be generated on its preceding server. These secrets must be transferred to the new server in a forward secure manner. Simply using communication over the SSL protocol will not work because SSL is not forward secure: SSL session keys can be recomputed from long-term secrets held by the receiver and recorded network traffic. So, if the receiving server is broken into at a later time period, it may still be possible for the attacker to retrieve the agent's secrets and break its forward security. In addition to being forward secure, the protocol must handle protocol bundle aborts in a manner that preserves forward security. Section 4.7.3 which describes the SDR key transfer protocol will address these issues. In the next section, we will briefly review the SDR forward secure partial result logging scheme to make its interaction with the key transfer protocol clear.

4.7.2 Result Logging Scheme

The SDR result logging scheme is simple: each log entry contains some data (e.g. some partial results) and a tag. There is also some auxiliary information associated with each entry: a sequence number denoting its position in the log and an end-of-period flag. The end-of-period flag is set for the final log entry that marks (attempted) migration out of that server. The tag in a log entry is generated by applying the selected cryptography scheme on the data and the auxiliary information.

4.7.3 Key Transfer Protocol

In order to prevent loss of forward security during transfer of an agent's cryptographic keys, we need to use a key transfer protocol that is forward secure. To ensure this, the SDR key transfer protocol uses an initial Diffie-Hellman key exchange phase to set up an ephemeral shared secret key instead of using SSL session keys to encrypt the communication. The ephemeral secret key is used to encrypt the forward secure keys.[1] The encrypted keys are transferred over an SSL channel to provide authenticated communication.

The SDR key transfer protocol runs in parallel with other agent migration protocols in the protocol bundle and its success or failure depends upon the successful completion of the other protocols. The protocol proceeds independently until the final (commit/abort) phase, when the status of the other protocols in the protocol bundle is checked. If the protocol bundle completes successfully, the SDR key transfer protocol initiates successful commit actions. This includes erasing the forward secure keys that were sent across to the next server. In the MAC-based forward security scheme, when a single agent is distributing its keys to multiple spawned agents, the agent gets a partial success bit vector that says which keys were successfully

[1] This step is also carried out in native code.

transmitted and which transfers failed. The agent has the freedom to redistribute its keys to any other agents it may spawn to complete its task. If the protocol bundle fails to complete successfully, the SDR key transfer protocol must initiate steps to gracefully abort the transaction even if the forward security keys were successfully transferred to the next server. The abort actions involve invalidation of the sent forward security key and generating a new log entry that signals a failed migration. Ideally, the agent should not reuse the keys already sent across in the aborted protocol. This is possible in the MAC-based forward security scheme by deriving two independent keys from the old one. If the protocol bundle aborts, the first key is invalidated and the second key is used to tag the "migration failed" log entry. However, in case of the signature-based forward security scheme, there is only one way to derive new keys from old ones. The newly derived key is used to tag the "migration failed" log entry and the agent can try to migrate again by deriving further from this key. Note however, that the result log is rendered potentially insecure from that time period onwards. If the server to which migration failed is malicious, it can try to subvert the computation by generating a fake result history for the agent.

4.8 Security-Aware Migration Itineraries

An agent may trust the different hosts that it wants to visit for accessing resources to differing degrees. Some hosts may be highly trusted by it, while others less so. The agent's security policy and the security attributes information it has about a host can aid it in determining a "trust metric" for that host.[2] Hosts with a sufficiently high trust metric are appropriate locations for verifying the integrity of the agent's partial results and securely planning future portions of its itinerary.[3] Trust metrics of the hosts that need to be visited can be one kind of input used in the itinerary planning process.

4.8.1 Using Trusted Hosts

The forward secure logging scheme allows successful detection of result tampering if there is only one malicious host on the agent's itinerary. Trusted hosts can be visited for verifying partial result integrity at different points in the itinerary to attain some degree of confidence in the final results of the computation. An agent must plan its itinerary such that trusted hosts are present at appropriate points in the itinerary. The cryptographic scheme used for protecting the result logs may restrict the placement of trusted verifi-

[2]How to determine the trust metric is an orthogonal issue, which is being investigated by the security metrics research community [23]. Our approach is not dependent upon any specific methodology. As long as a convenient method is available to determine the metric from available security data, agents can use it as a measure for the trustworthiness of the host.

[3]The itinerary is the actual sequence in which the agent will visit the hosts to access resources.

cation hosts. For example, consider a partial result log protected using the MAC-based forward security scheme. The agent wishes to visit a subset of the hosts on its itinerary and then verify the integrity of the generated results before proceeding to other hosts. The agent can follow a loop-like migration pattern in which it visits the subset of hosts sequentially with a given trusted host at the beginning and end of its trip. The agent must visit the trusted host at the beginning of its trip to save its forward secure verification secrets on it and after returning at the end of the trip it can use the saved secrets to verify the integrity of the generated results.

Apart from verification of the integrity of results, there are other security-related reasons to plan an itinerary with judicious placement of trusted hosts:

- **Prevention of collusion attacks:** Placing trusted hosts such that two untrusted (or insufficiently trusted) hosts are separated on the agent's itinerary by a trusted host can be used to attain some guarantee against the occurrence of collusion attacks on the forward security of the partial results.

- **Verification for long running agents:** Agents that run over long periods of time and cannot periodically migrate or send results back to their home server may use trusted hosts that are closer to periodically check the veracity of their results.

- **Early tamper detection:** An agent can arrange servers in short loops and migrate to a trusted host at the end of each loop if it does not sufficiently trust that the servers on the loop are honest. This focusses tamper detection over smaller sets of servers, possibly trading off task-completion time for a gain in security.

- **Correct execution on later servers:** If results generated on previous servers are required for correct computation on later servers, the agent should verify the correctness of its results as soon as it can. This prevents wastage of effort in computing the new results, in case the previous results have been tampered with.

- **Log compression:** An agent can migrate to a trusted host after visiting a number of other hosts. If the computed results verify correctly, then the agent may compress its result log in order to limit its main memory and migration bandwidth requirements.

4.8.2 Planning Secure Itineraries

An agent may use trusted intermediate hosts as bases for securely planning its itinerary in a piece-wise manner. At each trusted host, the agent decides what hosts it should visit next and in what order. Its decisions may be based on input from external modules that provide information such as the location of the resources it desires to use, how secure the host providing a specific resource is perceived to be (its trust metric) and the anticipated cost of execution at that host. Planning an itinerary or portion of an itinerary may be modelled as an optimization problem where the agent wishes to satisfy several constraints on characteristics like confidence in the results generated, cost of execution and migration latency. For instance, an agent may want to minimize the overall migration latency on its itinerary to achieve faster time to completion, while keeping the total cost of execution below some threshold.

In the general scenario, the constraints in the itinerary planning problem may be of several different forms. They can be equalities, inequalities or partial/total ordering constraints. A single constraint by itself—minimizing overall latency—is an instance of the NP-complete Travelling Salesman Problem (TSP). We will consider planning of secure itineraries as a sub-problem of the overall optimization problem which involves minimizing overall migration latency of the itinerary and keeping the confidence in the generated results above some reasonable, agent policy-specified threshold.

Agents may use security data about hosts providing the needed resources in different ways depending on its security policy. An agent that does not require a high level of security may simply ignore the security data. Another way in which the data may be used is to filter it through simple thresholds. For example, itinerary planning for agents in military applications may filter out all hosts that are below a certain trust metric (like the "top secret" clearance level) and all other hosts may be treated equally. In both of the above cases, we still have to minimize overall latency. If the agent's policy specifies a cutoff threshold for the overall latency below which it is acceptable, the problem is simplified. In such a case, it is possible to find a solution by making use of heuristic search techniques.[4]

4.9 Interposition Agents in Sanctuary

To ensure the separation of agents in our Java-based server, the servers load each agent into its own private class loader. Agents are prevented from obtaining references to objects owned by other agents, and thus can only communicate with each other through the server-provided communication ports.

[4] This is work in progress.

Interposition agents, as configured by the agent owner, are instantiated by the server before initiating a mobile agent. The interposition agent is transparently inserted on the ports between the mobile agent and the agent server. This allows the interposition agent to intercept all communications into and out of the agent, and to make modifications to these communications as needed. Interposition agents are written to the RPC interface, and examine or modify data on the communications link between the client and server. This allows interposition agents to implement the breadth of policy implementations possible in the capability-based protection model. The interposition agents in our design are similar in function to filters applied to capabilities [13], yet we do not require the introduction of a separate interface definition language. Interposition agents may also be imposed by the server for service agents, to provide strict access controls in a modular fashion.

Rights assigned to an agent are actually provided to the interposition agent, as it appears that the interposition agent is the one performing the accesses to the services on the agent server. In this way, interposition agents are used for rights elevation for an agent, allowing an essentially unprivileged agent to perform privileged actions. This contrasts our design with that of Jones [16], where interposition can only be used for restriction or semantics extensions, not rights elevation.

When the interposition agents are used for access restrictions, it can dynamically determine the severity of the restriction by monitoring the requests of the interposed agent. This would work in much the same way that the privileged monitoring process in Provos' privilege separation [25] dynamically decides which privileged requests should be accepted by modeling the unprivileged process with a finite state machine. Simpler static restrictions are more general, however, in that no analysis of the internal states of the interposed agents is needed.

Interposition agents can be used for restricting access to delegated rights using programmable (and stateful) access checks, interposing between a collection of mutually distrustful agents, applying an external policy language interpreter to actions taken by an agent and implementing a mandatory log of all actions taken by an agent. We will discuss the first two of these in the rest of this section.

4.9.1 Interposition Hierarchies

By separating the interposition concept from the agent structure itself and implementing it with the communications interface, it becomes possible to provide complex interposition hierarchies. When providing interposition on calls in the JavaSeal system, each interposition can only be performed on agents running below it in the seal hierarchy, as shown in Figure 4.

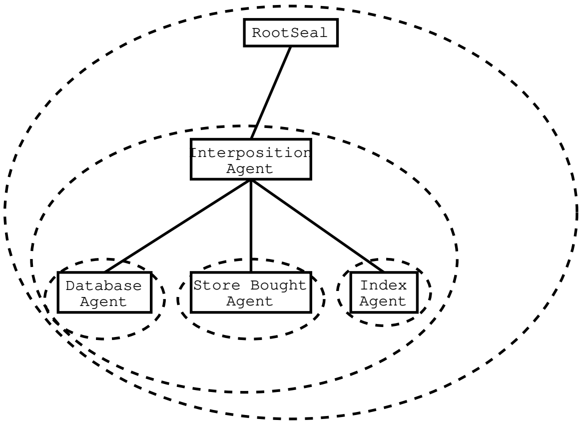

Figure 4: Multiple Interposition in JavaSeal

Interposition is possible for child seals.

Unfortunately, the example application shown in Figure 5 requires configurable interposition in a more complex manner. The store-bought, database and index agents are not fully trusted by the agent owner, though all are required to achieve an efficient solution to the indexing query that the agent owner wishes to have solved. The agent owner uses the interposition agent to control the communications behavior of the other three group members. In order to allow the database agent to exercise its privileges with the database resource, however, it must be allowed to communicate directly with the database. The interposition agent is configured to allow this, and provides the port to the database resource directly to the database agent, without interposing on it.

The three untrusted member agents in this example appear to be running in a normal, unrestricted, agent environment. Each agent communicates with the others as it would if it were not part of an interposed agent group, but rather were installed directly on the server.

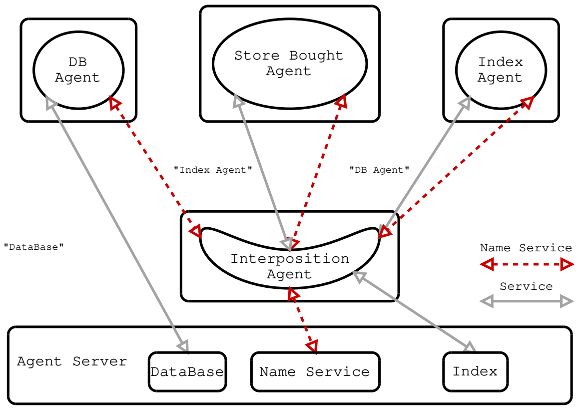

Figure 5: Multiple Interposition in Sanctuary

A single Interposition Agent can interpose between multiple other agents.

The same mechanism can be used for interposition on a service agent, as well. Service agent owners may similarly be untrusting of the service agents that they run, and use interposition agents to ensure that the communication patterns and services provided match the service agent's configuration.

4.9.2 Restricted Active Delegation via Interposition

In addition to simple policy checks to determine whether or not a particular access (or port communication) should be allowed, an interposition agent can be used for Restricted Active Delegation (RAD) of rights. Because code is transported to each server as part of the agent migration, adding additional code for performing restrictions on the agent will not add significantly to the complexity or communications requirements of the system.

Enforcing access control via RAD is strictly more powerful than static checks using delegation through certificates. Both delegation certificates and interposition code can be removed by malicious agent servers. The two are in this way equivalent.

Unfortunately, interposition is not a cure-all for enforcing policy decisions. In particular, operations whose typical behavior is success and which have a high retry overhead will be inefficient to control via interposition. Agent migration requires a large amount of preparation on the behalf of the agent, and thus would be a poor match with interposition as a policy enforcement mechanism. Rather than using a trial-and-error approach to find the next acceptable migration target (as is required when using interposition agents that simply deny the migration request), it would be better to use a policy that can be queried to determine which hosts will be acceptable migration targets.

5 Conclusion

We have described our security goals for a general-purpose mobile code system and discussed our approach to achieving these goals. In some cases, we have opted for the ability to detect compromises when preventing attacks is impractical or impossible. And when detection is difficult, we rely on trusted external information—in SACs—to provide a last line of defense.

The development of Sanctuary is on-going, and much more research remains to be done. For example, Mojo, our precompiler, is being extended to offer greater functionality. To fully explore the limits of mobile agent systems and their security properties, we need to develop and integrate additional security techniques and to test the system via deployment.

The usefulness of SACI is currently limited. There are a few public, independent security evaluation standards available [37, 24], but they are neither universally applicable nor widely used. Objective and practical security metrics are sorely needed; the nascent computer security insurance industry may help to improve their development [32].

Whether providing stronger security guarantees is practical remains an open question. New, more efficient forward secure signature schemes is an active area of investigation. Though it has been shown that program obfuscation is impossible in general [3], perhaps the class of programs that remain obfuscatable is still large enough to be of interest, permitting an efficient way to provide confidentiality of computation.

References

[1] Mike Accetta, Robert V. Baron, William Bolosky, David B. Golub, Richard F. Rashid, Avadis Tevanian, Jr., and Michael Wayne Young. Mach: A new kernel foundation for Unix development. In *Proceedings of Summer USENIX*, July 1986.

[2] Ross Anderson. Two remarks on public key cryptology. Unpublished. Available from `http://www.cl.cam.ac.uk/users/rja14/`.

[3] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In *CRYPTO2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, 2001.

[4] Mihir Bellare and Sara Miner. A forward-secure digital signature scheme. In M Wiener, editor, *CRYPTO99*, volume 1666 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[5] Mihir Bellare and Bennet Yee. Forward security in private key cryptography. Technical Report 2001/035, Cryptology ePrint Archive, 2001. Available at `http://eprint.iacr.org/2001/035/`.

[6] Matt Blaze, Joan Feigenbaum, John Ioannidis, and Angelos D. Keromytis. The KeyNote trust-management system, version 2. Internet RFC 2704, September 1999.

[7] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *Proceedings 1996 IEEE Symposium on Security and Privacy*, May 1996.

[8] Ciaran Bryce and Jan Vitek. The JavaSeal mobile agent kernel. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, Palm Springs, CA, USA, 1999.

[9] Yang-Hua Chu, Joan Feigenbaum, Brian LaMacchia, Paul Resnick, and Martin Strauss. REFEREE: Trust management for web applications. In *Proceedings of the 6th International World Wide Web Conference*, pages 227–238, 1997.

[10] Edward Elliott. Design and implementation of Mojo, a mobile agent precompiler. Master's thesis, University of California, San Diego, June 2000.

[11] Carl Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian Thomas, and Tatu Ylonen. SPKI certificate theory. Internet RFC 2693, September 1999.

[12] W. M. Farmer, J. D. Guttman, and Vipin Swarup. Security for mobile agents: Authentication and state appraisal. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, pages 118–130, September 1996.

[13] Daniel Hagimont and Leila Ismail. A protection scheme for mobile agents on java. In *Mobile Computing and Networking*, pages 215–222, 1997.

[14] Colin G. Harrison, David M. Chess, and Aaron Kershenbaum. Mobile agents: Are they a good idea? *Lecture Notes in Computer Science*, 1222:25–47, 1997.

[15] Matthew Hohlfeld. PhD thesis (draft). University of California, San Diego.

[16] Michael B. Jones. Interposition agents: transparently interposing user code at the system interface. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 80–93. ACM Press, 1993.

[17] G. Karjoth, D.B. Lange, and M. Oshima. A security model for aglets. *IEEE Internet Computing*, 1(4):68–77, 1997.

[18] Danny B. Lange and Mitsuru Oshima. *Programming & Deploying Mobile Agents with Java Aglets*. Addison-Wesley Co, 1998.

[19] N. A. Lynch. Concurrency control for resilient nested transactions. In *Proc. 2nd ACM Symposium on Principles of Database Systems*, March 1983.

[20] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997. ISBN 0-8493-8523-7.

[21] J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, 1985.

[22] Joon S. Park and Ravi S. Sandhu. Binding identities and attributes using digitally signed certificates. In *Proc. 16th Annual Computer Security Applications Conference (ACSAC)*, pages 120–127. Applied Computer Security Associates and ACM SIGSAC, December 2000.

[23] System Security Engineering Capability Maturity Model Project. *System Security Engineering Capability Maturity Model 2.0*, 1999.

[24] System Security Engineering Capability Maturity Model Project. *System Security Engineering Capability Maturity Model Appraisal Method Version 2.0*, 1999.

[25] Niels Provos. Preventing privilege escalation. Technical Report 02-2, Center for Information Technology Integration, University of Michigan, Ann Arbor, MI 48103-4943, August 2002.

[26] Richard F. Rashid. Threads of a new system. *Unix Review*, 4(8):37–49, August 1986.

[27] Ronald L. Rivest. Can we eliminate certificate revocations lists? In *Proc. Financial Cryptography 1998*, pages 178–183, 1998.

[28] Ronald L. Rivest and Butler Lampson. SDSI–a simple distributed security infrastructure. (see SDSI web page at `http://theory.lcs.mit.edu/~cis/sdsi.html`).

[29] Volker Roth. On the robustness of some cryptographic protocols for mobile agent protection. In *Proc. Mobile Agents 2001*, volume 2240 of *Lecture Notes in Computer Science*. Springer Verlag, December 2001.

[30] Volker Roth. Über die Bedeutung eines statischen Kernes für die Sicherheit Mobiler Software-Agenten. In *Kommunikationssicherheit im Zeichen des Internet*, DuD Fachbeiträge, pages 227–234. Vieweg-Verlag, March 2001.

[31] Volker Roth and Mehrdad Jalali. Concepts and architecture of a security-centric mobile agent server. In *Proc. Fifth International Symposium on Autonomous Decentralized Systems (ISADS 2001)*, pages 435–442, Dallas, Texas, U.S.A., March 2001. IEEE Computer Society. ISBN 0-7695-1065-5.

[32] Alex Salkever. E-insurance for the digital age. *Business Week Online*, April 2002.

[33] National Space and Aeronautics Administration. Pathfinder FAQ, 1997. `http://lunar.ksc.nasa.gov/mars/rovercom/rovfaqt.html#faq8`.

[34] Alfred Z. Spector and Michael L. Kazar. Wide area file service and the AFS experimental system. *Unix Review*, 7(3), March 1989.

[35] Sun Microsystems, Incorporated. Cache file system (cachefs). White paper, 1994.

[36] Trusted Information Systems. Trusted mach: Philosophy of protection, October 1996. Document No: TIS TMACH Edoc-0003-96A.

[37] U. S. National Institute of Standards and Technology. Federal information processing standards publication 140-1: Security requirements for cryptographic modules, January 1994.

[38] Bennet Yee. Monotonicity and partial results protection for mobile agents. Submitted to ICDCS, September 2002.

[39] Bennet S. Yee. A sanctuary for mobile agents. Technical Report CS97-537, University of California at San Diego, La Jolla, CA, April 1997. An earlier version of this paper appeared at the DARPA Workshop on Foundations for Secure Mobile Code.

[40] Bennet S. Yee. A sanctuary for mobile agents. In *Secure Internet Programming*, number 1603 in Lecture Notes in Computer Science, pages 261–274. Springer-Verlag Inc., New York, NY, USA, 1999.